What is Agile Development?

- Software equivalent of Lean Manufacturing
  - AKA Just-In-Time Manufacturing
- A family of development process
  - Have different key practices
  - Share common values
- Common Values
  - Incremental Delivery of
  - Working Software via
  - Collaboration rather than conflict

Why Not Waterfall?

- Only get one chance to get it right
- If you mess up the early part, the end can kill you
- You don’t find out whether you will survive until you get to the bottom (end)

Waterfalls are High Risk!
Continuous Delivery of Value
• New functionality is being delivered continuously
• Software is being modified continuously
• Changes can happen to any/every component
  – Components are not “frozen”
  – Can enhance any functionality

Practical Implications of Iterative Dev’t
• New functionality needs to be tested
  – New tests are required each iteration
• Bugs could be introduced anywhere,
  – regression testing is required of all existing components

Executing tests manually rapidly becomes the bottleneck!

The Solution: Automated Testing
• The obvious solution is to automate testing
  – At minimum, the regression testing
  – Ideally, the new functionality tests, too
• Two Kinds of Automated Tests:
  – Story-Tests describe requirements
  – Unit Tests improve code coverage

Recorded Test
• Exercise the application recording all the steps
• Replay the recorded test using the test runner
• Examples: Rational Robot, Mercury QTP

Traditional approach to automating tests
Why Recorded Tests?
- Relatively easy to record new tests
- Can often be done by non-technical people
- In theory, doesn’t take much longer to record than to run tests manually.

The Reality of Recorded Tests
- Hard to exercise all code paths via UI
  - Due to lack of built-in testability of application
- Recorded Tests are brittle and obscure intent
  - Very detailed, low-level recording
  - Typically via UI, an i/f designed for users not computers
- Tests are slow to execute
  - Due to asynchronous interactions with the application
- Tests can only be prepared after code is working
  - Useful as regression but not as definition of “Done”

The Reality of Recorded Tests (2)
- New functionality affects existing tests
  - UI may change
  - Behavior behind UI may change
  - Regression tests need to be updated regularly

In Practice, need to rerecord all the tests after every significant change to the application

Record, Refactor, Playback
- Record monolithic test
- Chop up recordings into small, intent-based scripts
- Hand-script the test cases using intent-based scripts as the steps.
- Like recording macros in Excel & then generalizing them.

Negates most benefits of Recorded Test

3 Test Automation Strategy Patterns
- Recorded Test
  - The traditional way
- Data-Driven Test
  - Using data files to drive tests (AKA Keyword-based)
- Scripted Test
  - Hand-written test programs

Data-Driven Test
- Encode tests as tabular data
- Write a test data interpreter to read data and exercise SUT
- Examples: FIT

- Main Advantage: Understandable by customer
- Main Issues: Limited vocabulary, separate from code
- Examples: FIT
Scripted Test

- Hand script tests against the SUT
- Execute tests in Test Runner
- Examples: xUnit family

Why Scripted Tests?

- Can become part of development process
  - Test-Driven Development

Why We Hand-Script Tests

- Self-Testing Code helps us:
  - Produce better quality software
  - Produce the right software
  - Work faster
  - Respond to change (agility)
- It does this by:
  - Providing focus
  - Providing rapid feedback
  - Reducing stress levels (anxiety)
- Applies the full power of programming language
  - Same language as production code
  - Full access to all the code to be tested

Testing Terminology

- Test vs SUT vs DOC:
  - Test
  - System Under Test
  - Depended on Component

- Unit vs Component vs Customer Testing

  - Black Box vs White Box
    - Black box: know what it should do
    - White box: know how it is built inside

Goals of Automated Developer Tests

- Before code is written
  - Tests as Specification
- After code is written
  - Tests as Documentation
  - Tests as Safety Net (Bug Repellent)
  - Defect Localization (Minimize Debugging)
- Minimize Cost of Running Tests
  - Fully Automated Tests
  - Repeatable Tests
  - Robust Tests

Intro to xUnit

- "Never in the field of software development have so many owed so much to so few lines of code."
- "The reason that JUnit is important, and deserves the Churchillian knock-off, is that the presence of this tiny tool has been essential to a fundamental shift for many programmers: Testing has moved to a front and central part of programming."

From Martin Fowler's foreword to "xUnit Test Patterns & Refactoring Test Code"
What is xUnit?

- A family of test automation frameworks
- One or more members for each programming language
  - JUnit, TestNG for Java
  - NUnit, MbUnit, VbUnit for .Net
  - TestUnit for Ruby
  - CppUnit, CppUnitLite for C++
  - PyUnit for Python
  - AbapUnit for ABAP
- Many are ports of JUnit to another language
- JUnit was a port of SUnit from SmallTalk

Moving Parts of xUnit

Tests are encoded as Test Methods consisting of 4 parts:
- Set up the preconditions ("fixture" or "context")
- Exercise the SUT by calling method(s)
- Verify the expected outcome has actually occurred
  - post-test state of SUT and DOCs
  - behavior during test (outgoing method calls)
- Clean up

Test Methods are put on Testcase Classes
- Tests are run using Test Runner

Typical Test Method

```java
public void testAllocationProfiledSitesAllHavingSettlements() throws Exception {
  //** Set up Scenario (Pre-conditions)  **//
  Retailer retailer = createAnonymousRetailer();
  Site aSite = createAnonymousSite();
  Wsi aWsi = createAnonymousWsi();
  aSite.assignRetailerEffectiveOn(retailer, intervalStartThisYear);

  //** Exercise System **//
  aWsi.allocateUnclaimedConsumption();

  //** Verify side effects (Post-conditions) **//
  List siteSettlements = IntervalSettlementHome.findAllForSiteAndInterval(aSite, allocationInterval);
  assertEquals("# of settlements in interval", 1, siteSettlements.size());
}
```

Typical Testcase Class

```java
public class TestInvoiceLineItems extends TestCase {
  TestAddItemQuantity_oneItem {...}
  TestAddItemQuantity_severalItems {...}
  TestAddItemQuantity_duplicateProduct {...}
  TestAddItemQuantity_zeroQuantity {...}
  TestAddItemQuantity_severalQuantity {...}
  TestRemoveItem_noItemsLeft {...}
  TestRemoveItem_oneItemLeft {...}
  TestRemoveItem SeveralItemsLeft {...}
}
```

Using xUnit for Test-Driven Dev't

- Rhythm: Red-Green-Refactor
- Red
  - Write a test that fails
  - Definition of “What done looks like”
  - “Pulls” code into existence via failing tests
- Green:
  - Write code to make the test pass
- Refactor:
  - Turn “code that works” into “clean code”
  - Continuous design improvement
**Outline**

- Introduction
  - Motivation
    - Why is Test maintainability important?
    - How do we make tests maintainable?
- Intro to Smells & Patterns
- Code Smells & Remedies
- Behavior Smells & Remedies
- Project Smells & Remedies
- Wrap Up

**What Does it Take To be Successful?**

Programming Experience
- xUnit Experience
- + Testing experience

Robust Automated Tests

---

**A Sobering Thought**

Expect to have just as much test code as production code!

The Challenge: How To Prevent Doubling Cost of Software Maintenance?

**Why are They so Crucial?**

- Tests need to be maintained along with rest of the software.
- Testware must be much easier to maintain than the software, otherwise:
  - It will slow you down
  - It will get left behind
  - Value drops to zero
  - You’ll go back to manual testing

Critical Success Factor:
Writing tests in a maintainable style

**Economics of Maintainability**

Test Automation is a lot easier to sell on
- Cost reduction than
- Software Quality Improvement or
- Quality of Life Improvement

Before Automation

<table>
<thead>
<tr>
<th>Development Effort</th>
<th>time</th>
</tr>
</thead>
</table>

After Automation

| Development Effort | time |

Initial Test Automation + Ongoing Maintenance

Initial effort

Increased effort

(Hump)

Ongoing effort

Index effort

Ongoing effort

Initial effort

Final effort
### Economics of Maintainability

Test Automation is a lot easier to sell on:
- Cost reduction than
- Software Quality Improvement or
- Quality of Life Improvement

![Graph showing Development Effort vs Test Automation Effort](chart.png)

#### Outline

- Introduction
- Motivation

#### Intro to Smells & Patterns
- What is a Test Smell?
- What is a Test Pattern?

#### Code Smells & Remedies
- Behavior Smells & Remedies
- Wrap Up

---

### What's a “Test Smell”?

- A set of symptoms of an underlying problem in test code
- Smells must pass the “Sniff Test”
  - A smell should be obvious
  - It should “grab you by the nose”
- Not necessarily the actual cause
  - There may be many possible causes for the symptom
  - Some root causes may contribute to several different smells

**Note:** Past literature often labels the cause as a smell. e.g. “Sensitive Equality” is really a cause of “Fragile Test”

### What's a “Pattern”?

- A “pattern” is a “recurring solution to a recurring problem”
  - E.g. A “Decorator” object lets us add behavior to a system dynamically by adding one or more decorators to an existing object.
- Must have been “invented” by three independent sources
  - That’s what makes it a “pattern” as in: “I see a pattern here!”
- The pattern exists whether or not it has been written up in the “pattern form”
  - Includes info on when (not) to use it

### What’s a “Test Pattern”?

- A “test pattern” is a recurring solution to a test automation problem
  - E.g. A “Mock Object” solves the problem of verifying the behavior of an object that should delegate behavior to other objects
- Test Patterns occur at many levels:
  - Test Automation Strategy Patterns
    - Recorded Test vs Scripted Test
  - Test Design Patterns
    - Implicit SetUp vs Delegated SetUp
  - Test Coding Patterns
    - Assertion Method, Creation Method
    - Language-specific Test Coding Idioms
    - Expected Exception Test, Constructor Test
Using Smells & Patterns

1. Write some tests
   • start with the easy ones!
2. Note the Test Smells that show up
3. Refactor to remove obvious Test Smells
   • Apply appropriate xUnit Test Patterns
4. Write some more tests
   • possibly more complex
5. Repeat from Step 2 until:
   • All necessary tests written
   • No smells remain

What's a Code Smell?

A problem visible when looking at test code:

• Tests are hard to understand
• Tests contain coding errors that may result in
  – Missed bugs
  – Erratic Tests
• Tests are difficult or impossible to write
  – No test API on SUT
  – Cannot control initial state of SUT
  – Cannot observe final state of SUT

• Sniff Test:
  – Problem must be visible (in their face) to test automater
  or test reader

Example

• Test addItemQuantity and removeLineItem methods of Invoice

<table>
<thead>
<tr>
<th>Customer</th>
<th>FirstName</th>
<th>LastName</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shipping</th>
<th>Billing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
</tr>
<tr>
<td>UnitPrice</td>
</tr>
<tr>
<td>ExtendedPrice</td>
</tr>
<tr>
<td>PercentDiscount</td>
</tr>
</tbody>
</table>

The Whole Test

```java
public void testAddItemQuantity_severalQuantity () throws Exception {
    try {
        // Setup Fixture
        final int QUANTITY = 5;
        Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30");
        Invoice invoice = new Invoice(customer);
        // Exercise SUT
        invoice.addItemQuantity(product, QUANTITY);
        // Verify Outcome
        List lineItems = invoice.getLineItems();
        if (lineItems.size() == 1) {
            LineItem actualLineItem = (LineItem)lineItems.get(0);
            assertEquals(invoice, actualLineItem.getInvoice());
            assertEquals(product, actualLineItem.getProduct());
            assertEquals(quantity, actualLineItem.getQuantity());
            assertEquals(new BigDecimal("30"), actualLineItem.getPercentDiscount());
            assertEquals(new BigDecimal("19.99"), actualLineItem.getUnitPrice());
            assertEquals(new BigDecimal("69.96"), actualLineItem.getExtendedPrice());
        } else {
            assertTrue("Invoice should have exactly one line item", false);
        }
    } finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
```
Verifying the Outcome

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"),
        actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"),
        actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"),
        actualLineItem.getExtendedPrice());
} else {
    assertTrue("Invoice should have exactly one line item",
        false);
}
```

Use Better Assertion

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"),
        actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"),
        actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"),
        actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}
```

Refactoring

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    assertEquals(invoice, actualLineItem.getInvoice());
    assertEquals(product, actualLineItem.getProduct());
    assertEquals(quantity, actualLineItem.getQuantity());
    assertEquals(new BigDecimal("30"),
        actualLineItem.getPercentDiscount());
    assertEquals(new BigDecimal("19.99"),
        actualLineItem.getUnitPrice());
    assertEquals(new BigDecimal("69.96"),
        actualLineItem.getExtendedPrice());
} else {
    fail("invoice should have exactly one line item");
}
```

Introduce Custom Assert

```java
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actualLineItem = (LineItem) lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
} else {
    fail("invoice should have exactly one line item");
}
```
The Whole Test

```java
public void testAddItemQuantity_severalQuantity () throws Exception {
    try {
        // Setup Fixture
        final int QUANTITY = 5;
        Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
        Invoice invoice = new Invoice(customer);
        // Exercise SUT
        invoice.addItemQuantity(product, QUANTITY);
        // Verify Outcome
        List lineItems = invoice.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actualLineItem = (LineItem) lineItems.get(0);
        LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
        assertLineItemsEqual(expectedLineItem, actualLineItem);
    }
    finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```

The Smells Seen Thus Far (2)

- **Other Obscure Test Causes:**
  - **Indirect Testing**
    - Interacting with the SUT via other software
  - **A cause of Fragile Tests (Behavior Smell)**
  - **Mystery Guest**
    - Lots of “Magic Numbers” or Strings used as keys to database.
    - “Lopsided” feel to tests (either Setup or Verification of outcome is external to test)

The Smells Seen Thus Far (3)

- **Conditional Test Logic**
  - Tests containing conditional logic (IF statements or loops)
  - Hard to verify correctness. Does it always test the same thing?
    - A cause of Buggy Tests (Project Smell)

- **Test Code Duplication**
  - Same code sequences appear many times in many tests
  - More code to modify when something changes
  - A cause of Fragile Tests (Behavior Smell)
The Patterns Used So Far

• Expected Objects
  – Use AssertEquals on whole objects rather than comparing individual fields

• Guard Assertions
  – Remove conditional logic associated with avoiding assertions when they would fail

• Custom Asserts
  – Remove Test Code Duplication by factoring out common code
  – Remove conditional logic associated with complex verification logic

The Whole Test

```java
public void testAddItemQuantity_severalQuantity () throws Exception {
    try {
        // Setup Fixture
        final int QUANTITY = 5;
        Address billingAddress = new Address("1234 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
        Customer customer = new Customer(99, "John", "Doe", new BigDecimal("3000.00");
        Invoice invoice = new Invoice(customer);

        // Exercise SUT
        invoice.addItemQuantity(product, QUANTITY);

        // Verify Outcome
        List lineItems = invoice.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actualLineItem = (LineItem) lineItems.get(0);
        LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
        assertLineItemsEqual(expectedLineItem, actualLineItem);
    } finally {
        deleteObject(expectedLineItem);
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```

Implicit Fixture Teardown - Naive

```java
public void tearDown() {
    deleteObject(expectedLineItem);
    deleteObject(invoice);
    deleteObject(product);
    deleteObject(customer);
    deleteObject(billingAddress);
    deleteObject(shippingAddress);
}
```

Implicit Fixture Teardown - Robust

```java
public void tearDown() {
    try {
        deleteObject(expectedLineItem);
    } finally {
        deleteObject(invoice);
    }
    try {
        deleteObject(product);
    } finally {
        deleteObject(customer);
    }
    try {
        deleteObject(billingAddress);
    } finally {
        deleteObject(shippingAddress);
    }
}
```
xUnit Test Patterns and Smells

**Pattern: Automated Fixture Teardown**

```java
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    addTestObject(shippingAddress);
}
```

```java
public void tearDown() {
    deleteAllTestObjects();
}
```

**Pattern: Transaction Rollback Teardown**

```java
public void setUp() {
    TransactionManager.beginTransaction();
}
```

```java
public void tearDown() {
    TransactionManager_abortTransaction();
}
```

*Important: SUT must not commit transaction

- DFT Pattern: Humble Transaction Controller

---

**The Smells Seen Thus Far**

- **Complex Undo Logic**
  - Complex fixture teardown code
  - More likely to leave test environment corrupted leading to Erratic Tests (Causes: Unrepeatable Tests or Interacting Tests)
The Patterns Used So Far

- **Inline Teardown**
  - Hand-coded tear down logic within the Test Method

- **Implicit Teardown**
  - Hand-coded tear down logic in a tearDown method

- **Automated Teardown**
  - Tear down all registered test objects programatically

- **Transaction Rollback Teardown**
  - Get the database to undo all the changes made by test
  - SUT must not commit transaction

---

The Whole Test

```java
public void testAddItemQuantity_severalQuantity () throws Exception {
    // Setup Fixture
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

---

Hard-Coded Test Data

```java
public void testAddItemQuantity_severalQuantity () {
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N 2V2", "Canada");
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"), billingAddress, shippingAddress);
    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

---

Distinct Generated Values

```java
public void testAddItemQuantity_severalQuantity () {
    final int QUANTITY = 5;
    Address billingAddress = new Address(getUniqueString(), getUniqueString(), getUniqueString(), getUniqueString(), getUniqueString());
    Address shippingAddress = new Address(getUniqueString(), getUniqueString(), getUniqueString(), getUniqueString(), getUniqueString());
    Customer customer = new Customer(getUniqueInt(), getUniqueString(), getUniqueString(), getUniqueDiscount(), billingAddress, shippingAddress);
    Product product = new Product(getUniqueInt(), getUniqueString(), getUniqueNumber());
    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

---

Creation Method

```java
public void testAddItemQuantity_severalQuantity () {
    final int QUANTITY = 5;
    Address billingAddress = createAnonymousAddress();
    Address shippingAddress = createAnonymousAddress();
    Customer customer = createCustomer( billingAddress, shippingAddress);
    Product product = createAnonymousProduct();
    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
Obscure Test - Irrelevant Information

Remove Irrelevant Information

Remove Irrelevant Information

Introduce Custom Assertion

The Whole Test – Done
xUnit Test Patterns and Smells

Test Coverage
Class TestInvoiceLineItems extends TestCase {
TestAddItemQuantity_oneItem {..}
TestAddItemQuantity_severalItems {..}
TestAddItemQuantity_duplicateProduct {..}
TestAddItemQuantity_zeroQuantity {..}
TestAddItemQuantity_severalQuantity {..}
TestAddItemQuantity_discountedPrice {..}
TestRemoveItem_noItemsLeft {..}
TestRemoveItem_oneItemLeft {..}
TestRemoveItem_severalItemsLeft {..}
}

Rapid Test Writing
public void testAddItemQuantity_severalItems () {
final int QUANTITY = 1 ;
Product product1 = createAnonymousProduct();
Product product2 = createAnonymousProduct();
Invoice invoice = createAnonymousInvoice();
// Exercise
invoice.addItemQuantity(product1, QUANTITY);
invoice.addItemQuantity(product2, QUANTITY);
// Verify
LineItem expectedLineItem1 = newLineItem(invoice,
product, QUANTITY, product.getPrice()*QUANTITY );
LineItem expectedLineItem2 = newLineItem(invoice,
product, QUANTITY, product.getPrice()*QUANTITY );
assertExactlyTwoLineItems(invoice,
expectedLineItem1, expectedLineItem2 );
}

The Smells Seen Thus Far
• Obscure Test
  The test is hard to understand. Specific causes:
  – Hard-Coded Test Data
    » Literal Constants
  – Irrelevant Information
    » Information in test unrelated to SUT behavior

The Patterns Used so Far
• Generated Value
  – Variation: Distinct Generated Value
    » Generate a unique value for each test run
• Creation Method
  – Anonymous Creation Method
    » Sets all attributes/references to default values
  – Parameterized Creation Method
    » Tests specifies relevant values only
• Testcase Class per Feature
  – Group all Test Methods for a feature or concept on a single class
  – Alternatives: Testcase Class per Class, Testcase Class per Fixture
• Custom Assertion
  → (again)

Hard to Test Code
• Code can be hard to test for a number of reasons:
  – Too closely coupled to other software
  – No interface provided to set state, observe state
  – Only asynchronous interfaces provided
• Root Cause is lack of Design for Testability
  – Comes naturally with Test-Driven Development
  – Must be retrofitted to legacy (test-less) software
• Temporary Workaround is Test Hook
  – Becomes Test Logic in Production (code smell) if not removed

Test Double Patterns
• Replace depended-on components with test-specific ones to isolate SUT
• Kinds of Test Doubles
  – Test Stubs return test-specific values
  – Test Spies record method calls and arguments for verification by Test Method
  – Mock Objects verify the method calls and arguments themselves
    » Fake Objects provide (apparently) same services in a “lighter” way
• Test Doubles need to be “installed”
  – Dependency Injection
  – Dependency Lookup
• Configurable Test Doubles are reusable but need to be configure with test-specific values
  – return values
    » expected method calls & arguments
### Testability Patterns

- **Humble Object**
  - Objects closely coupled to the environment should not do very much (be humble)
  - Should delegate real work to a context-independent testable object

- **Dependency Injection**
  - Client "injects" depended-on objects into SUT
  - Tests can pass a Test Double to control "indirect inputs" from dependents

- **Dependency Lookup**
  - SUT asks another object for its dependencies
  - Service Locator, Object Factory, Component Registry

- **Test-Specific Subclass**
  - Can extend the SUT to all access by test

### Recap of Code Smells

- **Conditional Test Logic**
- **Hard to Test Code**
- **Obscure Test**
- **Test Code Duplication**
- **Test Logic in Production**

### Recap of Patterns Used:

- **Expected Object**
- **Custom Assertion**
- **Guard Assertion**
- **Inline Teardown**
- **Implicit Teardown**
- **Automated Teardown**
- **Transaction Rollback Teardown**
- **(Anonymous/Parameterized) Creation Method**
- **(Distinct) Generated Value**
- **Humble Object**
- **Dependency Injection / Lookup**
- **Test Double / Test Stub / Mock Object / Fake Object**
- **Test-Specific Subclass**
- **Testcase Class per Feature/Fixture/Class**

### What's a Behavior Smell

- **A problem seen when running tests.**
- **Tests fail when they should pass**
  - or pass when they should fail (rarer)
- **The problem is with how tests are coded;**
  - not a problem in the SUT
- **Sniff Test:**
  - Detectable via compile or execution behavior of tests

### Common Behavior Smells

- **Slow Tests**
- **Erratic Tests**
  - Too many variants to list here
- **Fragile Tests**
  - The 4 sensitivities
- **Assertion Roulette**
- **Frequent Debugging**
- **Manual Intervention**
Agile Development Cycles

• Each cycle involves running the tests
• The tests must run “quickly enough”

Behavior Smell

Slow Tests

- Slow Tests
  - It takes several minutes to hours to run all the tests
- Impact
  - Lost productivity caused by waiting for tests
  - Lost quality due to running tests less frequently
- Causes:
  - Slow Component Usage
    - e.g. Database
  - Asynchronous Test
    - e.g. Delays or Waits
  - General Fixture
    - too much fixture being setup

Avoiding Slow Tests – Slow SUT

- Run Tests Faster
  - Get faster hardware
    - E.g. Quad-processor test execution box
- Avoid Slow Code
  - Avoid Fixture Persistence
    - Use a Fresh Fixture with Fake Database
  - Avoid slow components
    - Replace with Test Double (or Test Stub)
- Run Fewer Tests
  - Run subsets of tests when possible (e.g. pre-checkin)
  - Run all the tests sometime, somewhere! (e.g. overnight)

Avoiding Slow Tests – Slow Test Code

- Avoid Waits
  - Use Humble Object to avoid Asynchronous Test by testing logic directly
- Test Less Code
  - Reduce Test Overlap
- Set Up Less Fixtures
  - Use a Minimal Fixture
- Set Up Fixtures Less Often
  - Reuse a Shared Fixture

Shared Test Fixture

- What it is:
  - Improves test run times by reducing setup overhead.
  - A “standard” test environment applicable to all tests is built and the tests reuse the same fixture instance.
  - Setup
    - Exercise
    - Verify
    - Teardown
  - Fixtures
  - SUT

- Variations:
  - Fixture is shared between some/all the tests in a single test run
  - Fixture may be shared across many TestRunners (Global Test Fixture)
- Examples:
  - Standard Database contents
  - Standard Set of Directories and Files
  - Standard set of objects

Bad Smell Alert:

Erratic Tests
Setting Up the Shared Test Fixture

To share the same fixture instance between tests:

- **Prebuilt Fixture**
  - Fixture is built ahead of time and reused by many test runs
- **Lazy Setup**
  - First reference causes it to be initialized
  - How do you know when to clean up?
- **SuiteFixture Setup**
  - Use Static variables to hold the fixture
  - Initialize one before first test; destroy after last
- **Setup Decorator**
  - Define a Test Decorator that implements Test
  - Wrap the test suite with an instance of the decorator

To share the same fixture instance between tests:

- Use only when don't need to clean up the fixture
- Tests that depend on the decorator cannot be run without it.
- Only supported by NUnit, VBUnit, JUnit 4.0

Erratic Tests

- **Interacting Tests**
  - When one test fails, a bunch of other tests fail for no apparent reason because they depend on other tests' side effects
- **Unrepeatable Tests**
  - Tests can't be run repeatedly without intervention
- **Test Run War**
  - Seemingly random, transient test failures
  - Only occurs when several people testing simultaneously
- **Resource Optimism**
  - Tests depend on something in the environment that isn't available
- **Non-Deterministic Tests**
  - Tests depend on non-deterministic inputs

Erratic Tests – Interacting Tests

If many tests use same objects, tests can affect each other's results.

- Test 2 failure may leave Object X in state that causes Test n to fail.

Symptoms:

- Tests that work by themselves fail when run in a suite.
- Cascading errors caused by a single bug failing a single test.
- Bug need not affect other tests directly but leaves fixture in wrong state for subsequent tests to succeed.

Erratic Tests – Unrepeatable Tests

If many test runs use same objects, test runs can affect each other's results.

- Test 2 update may leave Object X in state that causes Test 1 to fail on next run.

Symptoms:

- First run after opening the TestRunner or re-initializing Shared Fixture behaves differently
  - Succeed, Fail, Fail
  - Fail, Succeed, Succeed
  - Resetting the fixture may "reset" things to square 1 (restoring the cycle)
  - Closing and reopening the test runner for in-memory fixture
  - Reinitializing the database

Erratic Tests – Test Run War

- If many test runners use the same objects (from Global Fixture), random results can occur.
  - Interleaving of tests from parallel runners makes determining cause very difficult

Erratic Tests – Non Deterministic Test

Tests depend on non-deterministic inputs.

Symptoms:

- Tests pass at some times; fail at other times
  - Lack of control over time/date when system contains time/date logic (addressed by getting control of indirect input via a stub)
  - Tests use different values in different runs
## Erratic Tests – Resource Optimism

Tests depend on non-ubiquitous external resources.  
**Symptoms:**
- Tests pass in some environments; fail in others
  - SUT depends on something in the environment that is not always present.
  - Addressed by creating it during the fixture setup phase

### Avoiding Erratic Tests - Fresh Fixture

**What it is:**
- "Brand new" fixture built for each test
- Tests are completely independent

**Variations:**
- Transient Fresh Fixture
  - Fixture automatically disappears at end of each test
  - e.g. Garbage-collected TearDown
- Persistent Fresh Fixture
  - Fixture naturally “hangs around” after test
  - Requires extra effort to ensure it is fresh

### Reducing Erratic Tests - Shared Fixture

- Avoid Interactions between Test Runners
  - Give each developer their own Database Sandbox.
  - Avoids Test Run Wars but not Interacting Tests, etc,
- Don’t Change Shared Fixture
  - Immutable Shared Fixture avoids Interacting Tests
  - Create Fresh Fixture for objects to be changed
    - (See Persistent Fresh Fixture)
  - Challenge: What constitutes a “change” to a fixture?
    - Change existing objects / rows -> YES!
    - Add new objects related to existing objects -> SOMETIMES!

### Build new Shared Fixture for each run
- Avoids Unrepeatable Tests
- When:
  - Lazy Setup
  - Setup Decorator
  - SuiteFixture Setup

## Fragile Tests

**Causes:**
- Interface Sensitivity
  - Every time you change the SUT, tests won't compile or start failing
  - You need to modify lots of tests to get things “Green” again
  - Greatly increases the cost of maintaining the system
- Behavior Sensitivity
  - Behavior of the SUT changes but it should not affect test outcome
  - Caused by being dependent on too much of the SUT’s behavior.
Fragile Tests (2)

Causes (continued):

- **Data Sensitivity**
  - Alias: Fragile Fixture
  - Tests start failing when a shared fixture is modified
    - e.g. New records are put into the database

- **Context Sensitivity**
  - Something outside the SUT changes
    - e.g. System time/date, contents of another application

Avoiding Interface Sensitivity

- **Use Stable Interfaces**
  - Bypass Presentation Layer (UI)
  - Backwards compatibility of changes to used interface
    - e.g. Facade

- **SUT API Encapsulation**
  - Hide non-essential parts of SUT API from Test Methods
    - Creation Method
    - Finder Method
    - Verification Method

Avoiding Data/Context Sensitivity

- **Minimal Fresh Fixture**
  - Use a Fresh Fixture
  - Custom design it for each test.
  - Avoid a Standard Fixture that could become a Fragile Fixture

- **Test Stubs**
  - Replace the need for real fixture by using a Test Stub to provide indirect inputs

Assertion Roulette

- **Symptom:**
  - One or more unit tests are failing in the automated build and you cannot tell why without rerunning the tests in your IDE. When you cannot reproduce the problem in your IDE you have no idea what is going wrong.

- **Impact:**
  - It takes longer to determine what is wrong with the code.
  - Bugs that cannot be reproduced cannot be fixed.

- **Root Cause:**
  - Missing/Unclear Assertion Messages

- **Solution:**
  - Use the right Assertion Method.
  - Add Assertion Messages to all Assertion Method calls
  - Write Diagnostic Custom Assertion

Diagnostic Custom Assertion

- **Variation of Custom Assertion**
- **Comparison:**
  - Compares its inputs in a way that provides useful diagnostic messages.
  - e.g. assertEquals does this:
    - expected <nil> but was <abc>
    - strings differ starting at position 247; expected `<...abcdefghi...>` but was `<...abcxyzghi...>`

Frequent Debugging

- **Symptom:**
  - One or more tests are failing and you cannot tell why without resorting to the debugger. This seems to be happening a lot lately!

- **Impact:**
  - Debugging is a very time-intensive activity.
  - While it may help you find the bug, it won’t keep it from coming back.

- **Root Causes:**
  - Missing Unit Tests
  - Poor Assertion Messages

- **Solution:**
  - Better unit test coverage of the code
  - More/Better Assertion Messages
Recap of Behavior Smells

- Slow Tests
- Erratic Tests
  - Too many variants to list here
- Fragile Tests
  - The 4 sensitivities
- Assertion Roulette
- Frequent Debugging
- Manual Intervention

Recap of Patterns

- Shared Fixture
- Fresh Fixture
- Standard Fixture
- Minimal Fixture
- Lazy Setup
- Setup Decorator
- SuiteFixture Setup

Outline

- Introduction
- Motivation
- Intro to Smells & Patterns
- Code Smells & Remedies
- Behavior Smells & Remedies
- Project Smells & Remedies
- Wrap Up

What’s a Project Smell?

- A Test Smell that a project manager is likely to observe
- Symptoms are typically developer behavior or feedback from other organizations
- There may be metrics that point out the smell
  - e.g. Number of bugs found in Acceptance Test
- Root cause is often Code or Behavior Smells
- Cannot be addressed directly
  - Solution is to address underlying smell(s)

Common Project Smells

- Developers Not Writing Tests
- Buggy Tests
- Production Bugs
- High Test Maintenance Cost

Developers Not Writing Tests

- Symptoms:
  - No tests can be found when you ask to see the unit tests for a task,
  - customer tests for a User Story,
  - Lack of clarity about what a user story or task really means
- Impact:
  - Lack of safety net
  - Lack of focus
- Possible Causes:
  - Don’t have the skills?
  - Have been told not to?
  - Not enough time?
  - Don’t see the value?
Buggy Tests
• Symptoms:
  – Tests are failing when they shouldn’t (the SUT works fine)
• Impact:
  – No one trusts the tests any more
• Possible Causes:
  – Erratic Tests
  – Fragile Tests
  – Untested Test Code
  – Untested Code

Production Bugs
• Symptoms:
  – Bugs are being found in production
• Impact:
  – Expensive trouble-shooting
  – Development team’s reputation is in jeopardy
• Possible Causes:
  – Lost/Missing Tests
  – Slow Tests
  – Untested Code
  – Hard-to-Test Code
  – Developers Not Writing Tests

High Test Maintenance Cost
• Symptoms:
  – A lot of effort is going into maintaining the tests
• Impact:
  – Cost of building functionality is increasing
  – People are agitating to abandon the automated test
• Possible Causes:
  – Erratic Test
  – Fragile Test
  – Buggy Test
  – Obscure Test
  – Hard to Test Code

Outline
• Introduction
• Motivation
• Intro to Smells & Patterns
• Code Smells & Remedies
• Behavior Smells & Remedies
• Project Smells & Remedies
  – Wrap Up

What Next?
• You have a better idea of:
  – what can be achieved
  – problems to look for
  – Test Smells
  – symptoms (smells) vs root causes
• You have an initial list of patterns to address root causes
  – More at the web site and in the book
• Time to go “Smell Hunting”

Be Pragmatic!
• Not all Smells can (or should) be eliminated
  – Cost of having smell vs. cost of removing it
  – Cost to remove it now vs. cost of removing it later
• Catalog of Smells and Causes gives us the tools to make the decision intelligently
  – Trouble-shooting flow chart
  – Suggested Patterns for removing cause
• Catalog of Patterns gives us the tools to eliminate the Smells when we choose to do so
  – How it Works
  – When to Use It
  – Before/After Code samples
  – Refactoring notes
What Does it Take To be Successful?

Programing Experience
+ xUnit Experience
+ Testing Experience
+ Design for Testability
- Test Smells
+ Test Automation Patterns
+ Fanatical Attention to Test Maintainability
= Robust, Maintainable Automated Tests

More on xUnit Patterns & Smells

- Book:
  xUnit Test Patterns
  Refactoring Test Code
  by: Gerard Meszaros
  available Now!
- Website:
  http://xunitpatterns.com
  With handy links to purchase

Thank You!
Gerard

Books on xUnit Test Automation
- xUnit Test Patterns – Refactoring Test Code
  – Gerard Meszaros
- Test-driven Development - A Practical Guide
  – David Astels
- Test-driven Development - By Example
  – Kent Beck
- Test-Driven Development in Microsoft .NET
  – James Newkirk, Alexei Vorontsov
- Unit Testing With Java - How tests drive the code
  – Johannes Link
- JUnit Recipes
  – J.B. Rainsberger

Other Useful Books
- Working Effectively with Legacy Code
  – Michael Feathers
- Fit for Software Development
  – Rick Mugridge, Ward Cunningham
- Refactoring - Improving the Design of Existing Code
  – Martin Fowler plus contributors
- Design Patterns: Reusable Elements of Design
  – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides